Test Execution Control Tool: Automating Testing in Spacecraft Integration and Test Environments

Michael Levesque
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
818-393-7987
Michael.Levesque@jpl.nasa.gov

John Louie
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
818-354-146
John.Louie@jpl.nasa.gov

Ana Guerrero
Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, CA 91109
818-354-1317
Ana.Guerrero@jpl.nasa.gov

Abstract—In the era of faster, better, cheaper, JPL aims to develop and integrate an ever increasing number of spacecraft subsystems and instruments. One of the problems JPL faces is to quickly, efficiently and accurately test these subsystems in a diverse set of environments from the breadboard, to environmental simulation and through launch preparation. The Test Execution Control Tool was developed using a Tool Command Language (Tcl) [1] based test script interpreter to aid engineers in automating flight system test procedures.

The Test Execution Control Tool (TECT) is a Tcl based application that allows testers to develop and regulate the execution of a test script in a spacecraft test environment. Using the TECT, a tester can make the diverse collections of ground software, ground support equipment and flight systems work together through automated scripts. Tcl is the language of choice because of its ease of use, breadth of functionality, extensibility, platform independence and open source. By adding a graphical user interface for controlling Tcl script execution and adding interfaces for simultaneously commanding and receiving telemetry from the test equipment and the system under test, a spacecraft tester is provided a single point of test execution control. Additionally, tests can be fully automated with closed loop control between command and telemetry from all test support equipment and telemetry analysis software.

The TECT is a proven tool in the mission critical testing environment bringing a project several advantages. It is easy to use, extensible and allows engineers to automate testing tasks to a much higher degree than they otherwise could. Because it allows projects to access and coordinate all the components of the complex testing environment, you not only achieve broader test coverage, leading to higher product assurance, but you achieve more speed and efficiency, leading to lower costs and faster results. Furthermore the tools extensibility and flexibility allow the mission to use the same ground processing software during test as the mission operator will use during flight.

TABLE OF CONTENTS

1. INTRODUCTION

The Jet Propulsion Laboratory supports unmanned deep space missions through an Advanced Multi-Mission Operations System (AMMOS) that is developed and operated under the Telecommunications Mission Operations Directorate (TMOD). The AMMOS supports all JPL deep space mission flight operations. It is also used during spacecraft integration and mission Acceptance Test and Launch Operations (ATLO) providing full mission life cycle support.

JPL and its partners plan to develop, test and operate more missions over shorter life cycles than in the past. Figure 1 illustrates the mission support roadmap [2].

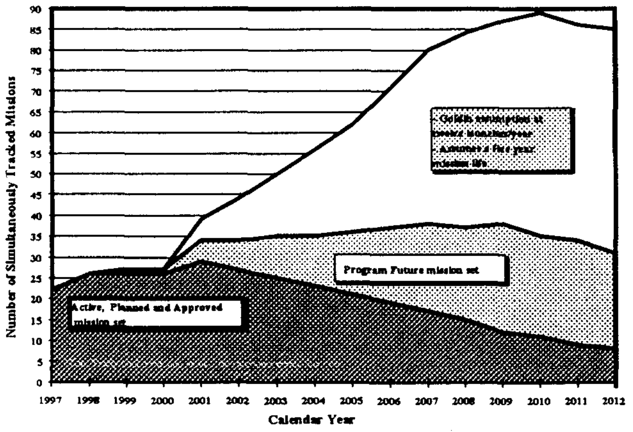


Figure 1. Mission Support Roadmap

In order to accomplish this, opportunities are being explored to shorten mission development life cycles while increasing quality standards. The Test Execution Control Tool (TECT) described in this paper is one approach to shortening spacecraft and instrument integration and test life cycle times and cost while increasing the quality of the test program.

The TECT is presented in the following three sections. The first section presents an overview of the Test Execution Control Tool functional description. The second part describes the TECT detailed design. The third part uses a case study to cover the operational aspects and results from using the TECT. Finally, a brief discussion of future work and conclusions are provided.

2. Functional Description

The primary objective of the TECT is to provide functionality for testing instrument or spacecraft systems. This section describes the TECT functionality.

Test scripting and control

The TECT provides an interpreted scripting language for specifying and controlling tests. The language is used to write test scripts to automate spacecraft or instrument integration, functional and performance tests. Primitive commands to spacecraft, instruments and test support equipment are bundled into test scripts. Control feedback from monitored data can also be added to the scripts. The TECT script interpreter reads these scripts, checks for syntax errors before execution and issues commands to the flight and test support subsystems in a specified time sequence. The TECT executes and synchronizes test sequences from one central point even if the subsystems which are under its control are distributed. Test sequences are reusable since selecting and controlling test sequences through TECT is flexible. The number of test sequences and control through TECT is virtually unlimited.

Interfaces to Distributed Processes

One principle function of the system test environment and TECT is to centralize testing by coordinating the operation of several distributed subsystems. Many of these distributed test activities occur simultaneously. These activities include[3];

a) Commanding of the flight instrument or spacecraft,
b) Control of the test support equipment,
c) Collection and recording of test data from the instrument or spacecraft and test support equipment,
d) Instrument or spacecraft and support data health checks,
e) Conversions and display for instrument or spacecraft and test support data engineering units,
f) Distributing test data for post test performance analysis.

The TECT integrates each of the distributed processing elements by providing a remote data monitor and control interface. Control with feedback from monitored data is provided through these interfaces for each subsystem involved in a test.

Event Reporting and Logging

Event reporting and logging are critical to test case analysis and trouble shooting. The TECT as three classes of logged data; that logged under user control, that logged from sending a command to one of the remote subsystems, and that logged as a result of a user initiated control event from the user interface. User logged events generally represent significant data taken during a test that is required for post test analysis. Logged commanding events are performed when commands are send to the instrument or spacecraft and to the test equipment. These generally aid in troubleshooting anomalies with flight software or hardware interfaces. Finally all user controls are logged to again troubleshoot anomalies with the test procedures themselves or provide status of the testing activity.

3. Design and Implementation

This section provides an overview of the end-to-end information system functional design. It establishes the environment in which the TECT is used. This is followed by a detailed description of the TECT design including discussion on the tradeoffs made during the design process.

Test System Function Design

The Test, Telemetry and Command Subsystem (TTACS) consists of software and hardware, which, combines operational Multi-mission Ground Data System (MGDS) applications, to provide a telemetry and command data system in spacecraft test [4]. TTACS components allow data to flow to and from the instrument or spacecraft and test support equipment. Components of the MGDS for test include flight and support equipment data acquisition, telemetry processing including frame synchronization, decoding, and channelization, telemetry data cataloging, archiving, retrieval and distribution, Engineering data processing for converting data numbers to engineering units, alarming and displaying engineering data, and various other tools for analyzing and displaying telemetry data as shown in Figure 2.

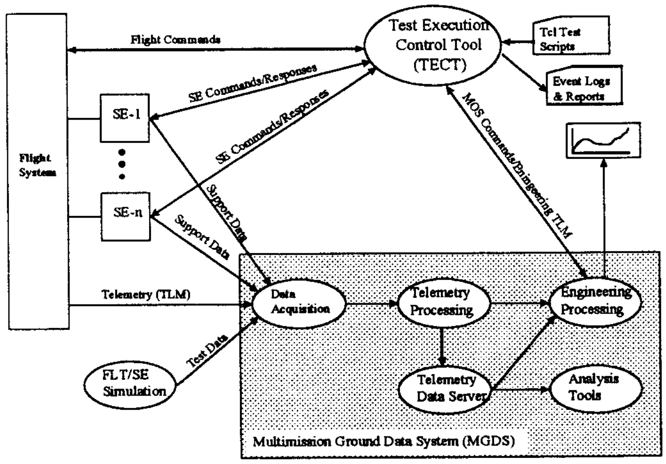


Figure 2. EEIS Design Diagram

Test Execution Control Tool Design

The TECT design includes three components, the script interpreter, the graphical user interface and the control interfaces as shown in Figure 3. The test script interpreter is based on the Tool Control Language (Tcl). Test scripts written in Tcl are input, syntax checked and executed by the TECT script interpreter. The graphical user interface, called builder, is used to control the flow of the executing test script. The builder also provides functions for debugging anomalies either in the test script itself or the system under test. Finally the control interfaces are used to send commands and receive responses from the distributed subsystems. These control interfaces isolate the subsystem specific interface protocols from the script interpreter.

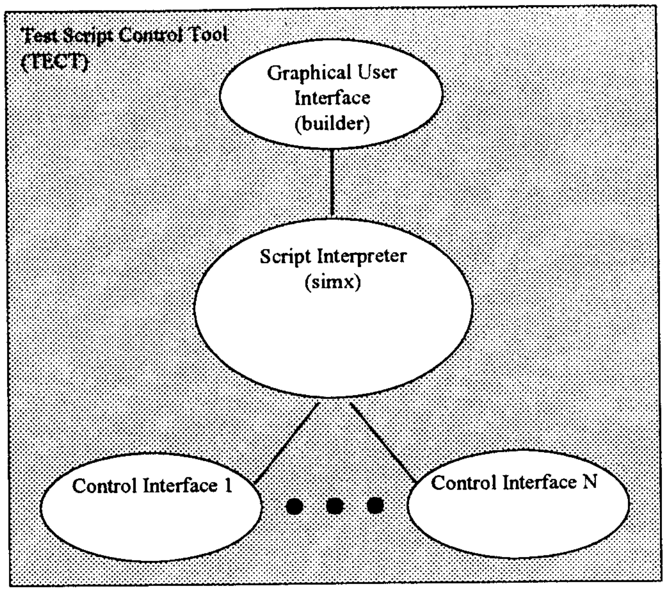

Test Script Control Tool (TECT)
Graphical User Interface (builder)
Script Interpreter (simx)
Control Interface 1
Control Interface N

Figure 3. TECT Detailed Design Diagram

Test Execution Control Tool Scripting Language

The TECT was built using a scripting language for several reasons [5];

a) the tests to be performed were not well defined and would change rapidly over time,
b) the main test of the application was to integrate and coordinate a set of existing components,
c) the application needed to manipulate a variety of different things,
d) it must have graphical user interface capabilities,
e) the application does a lot of string processing,
f) functionality would evolve over time,
g) it must be easy to extend and customize in the field.

Three scripting languages were considered; Perl, a language developed in-house and Tcl. The criteria for evaluating them included [6];

a) Widely used in the IT community,
b) In-house expertise,
c) Easy to use,
d) Easy to extend,
e) Support for graphical user interfaces,
f) Low cost or freely available.

Perl provided the largest IT and in-house community. Extensions are easily added and widely available from the user community. Support for graphical interfaces is not supported but can be added by integrating additional packages or third party software. It is freely available and supported on a wide variety of platforms. Additionally it performs well and it's implementation is robust. The one problem with Perl was its ease of use. It was generally considered too complex a language for the user community that would be responsible for developing and reviewing the test scripts.

A language developed in-house was consider since it would offer the development team the greatest control over the language and could be highly tailored to the integration and test community. But the in-house language would not be able to provide in the field extension. Since the test procedures and scripts were ill-defined during the TECT development phase, this lack of extension in the field was the greatest problem with an in-house language. Furthermore an in-house language would lack general IT and in-house community support. Finally it would lack graphical user interface support.

Tcl was the single most inviting of the languages. First it is very easy to use. It is highly extensible by both the development team and the test scripting team It has wide IT and in-house community support particularly in the test automation domain. Additionally, graphical user interfaces could be supported through the Tk [1] extension. Finally, it is freely distributed and available on at least as many platforms as Perl.

TECT Language Extensions

Extensions are modules written in a language different from Tcl which extend the functionality of Tcl in some way [7]. There were three reasons to extend Tcl for the TECT;

a) To add flight instrument test domain-specific functionality, in the form of the **procedure** command,
b) To interface the Tcl scripting language environment to existing software, in the form of the **route** command,
c) To provide a flexible way to access many test script modules, in the form of the **call** command.

The procedure command was added to designate high-level procedural text within the test script. Spacecraft and instrument system test involves the definition of well defined procedures that have high visibility for the test team. The functionality provided by the procedure command was to make this text easily identified through the graphical user interface and provide test script controls for it similar to those provided of the more primitive scripting commands.

The route command was added to provide a generic message oriented interface to the subsystem under control of the TECT. The route command has the form:

```
route <destination> <command>
```

Where destination is specified for a control interface and command is the string to be sent to that control interface destination. In Seawinds the control interfaces were for the flight system, the three subsystem support equipment and the telemetry processing software.

The call command was added to invoke a script module from within other script modules. The call command has the form:

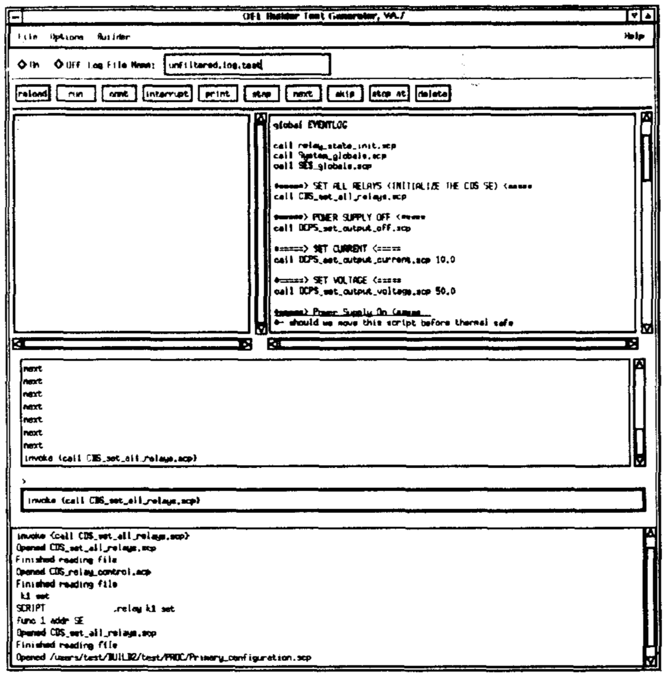
```
call <script name>
```

Where the script name is another script module that will be loaded and interpreted by the TECT given the current executing environment controls. The call was primarily used by test script writers to reuse test sequences in many different high-level procedures.

TECT Graphical User Interface and Test Script Control

The graphical user interface developed in Motif is shown in Figure 4. It communicates with the TECT script interpreter through a socket interface and starts the TECT interpreter. At run time help files are read and user defined Tcl built in commands are read from an input file. The reminder of this section describes the operation of the TECT user interface.

The interface window includes a menu bar, a row of control buttons, a row of user programmable macro buttons a display areas on the left for a test summary of the steps in the test procedure, a display areas on the right for the actual test script, a command message area, and a scrolling message area across the bottom of the window.



```
unfiltered.log.test
```

```
global EVENTLOG

call relay_state_init.scp
call System_globals.scp
call SES_globals.scp

#=====> SET ALL RELAYS (INITIALIZE THE CDS SE) <=====
call CDS_set_all_relays.scp

#=====> POWER SUPPLY OFF <=====
call DCPS_set_output_off.scp

#=====> SET CURRENT <=====
call DCPS_set_output_current.scp 10.0

#=====> SET VOLTAGE <=====
call DCPS_set_output_voltage.scp 50.0

#=====> Power Supply On <=====
#- should we move this script before thermal safe
```

```
next
next
next
next
next
next
next
invoke (call CDS_set_all_relays.scp)
```

```
>
invoke (call CDS_set_all_relays.scp)
```

```
invoke <call CDS_set_all_relays.scp>
Opened CDS_set_all_relays.scp
Finished reading file
Opened CDS_relay_control.scp
Finished reading file
 k1 set
SCRIPT              .relay k1 set
func 1 addr SE
Opened CDS_set_all_relays.scp
Finished reading file
Opened /users/test/BUILD2/test/PROC/Primary_configuration.scp
```

Figure 4. TECT Graphical User Interface

As each command in a script is executed, it is underlined on both the left and the right script display areas.

The **File** menu has **Open**, **Open Clear**, and **Exit** options. Open clear resets all predefined Tcl script variables.

The **Options** menu contains **Clear (Buffer) Modes, Show Command Area** toggle, and **Restart SIMX** (the Tcl Interpreter).

The **Builder** menu contains help files for mission specific flight commands.

The **Help** menu contains Tcl help (as opposed to interface help).

The row of test script control buttons under the menu bar includes the following:

- **reload** Reloads the current script and places the step at the top of the script.
- **run** Runs the loaded script file from beginning to end, or until a Stop-marked statement is reached.
- **cont** Runs the script continuously from the current step to the end or until a stop-marked statement is reached.
- **interrupt** Stops a running script at the current position.
- **print** Prints the value for the highlighted text. For example, if you highlight a variable in the text and click on the print button, the current value of that variable is printed in the message area of the display.
- **step** Performs the next step in a line of script. There may be more than one step in a statement; for example, the result of one calculation or variable value must be determined before the command can be completed. Message area reports"step."
- **next** Performs an entire command, which may include more than one operation. Message area reports "next."
- **skip** Skips a statement in a script.
- **stop at** Inserts a stop point. (Currently, the stop point is not marked on the interface.)
- **delete** Deletes a stop mark.
- **invoke** Copies highlighted text, preceded by the "invoke" command, to the command message area. Pressing <ENTER> then invokes the command. Meant primarily as a timesaver in typing commands.
- **goto** Moves to the highlighted step in the procedure, as displayed in the script syntax area (on the right). Then, clicking **cont, step,** or **next** executes the script from that point.

4. OPERATIONS

In this section, operational use of the TECT is described using a case study of the Seawinds Instrument Integration and Test team. The instrument integration and test team was responsible for system functional and performance testing of the instrument. The team used the TECT and TTACS shown in Figure 5 to perform this testing, including at the subsystem level, during subsystem integration, in environmental test and prior to integration with the spacecraft. Each of these environments required slightly different configurations requiring the TECT and TTACS test system to be flexible enough to accommodate missing flight system and support equipment functions. Additionally the test scripts and scripting language needed to support modularity so reuse could be maximized. This section describes the test scripts that were developed for Seawinds system integration and test.

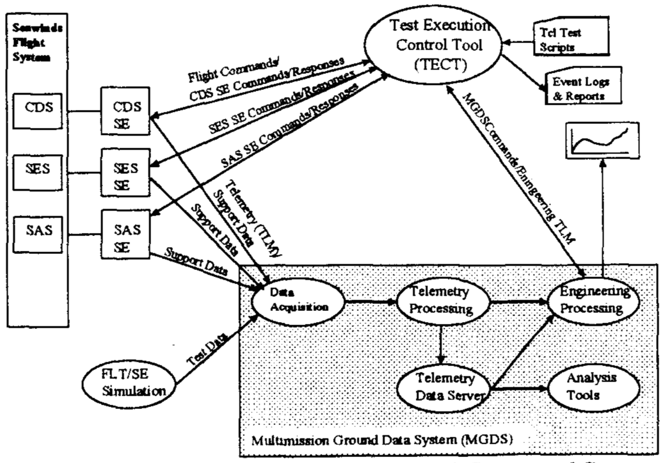


Figure 5. Seawinds Test, Telemetry and Command System

Test Script Design and Examples

By extending the Tcl within TECT to include the function **call**, the test and integration team layered the test scripts into sets of modules. This layering took the form shown in Figure 6.

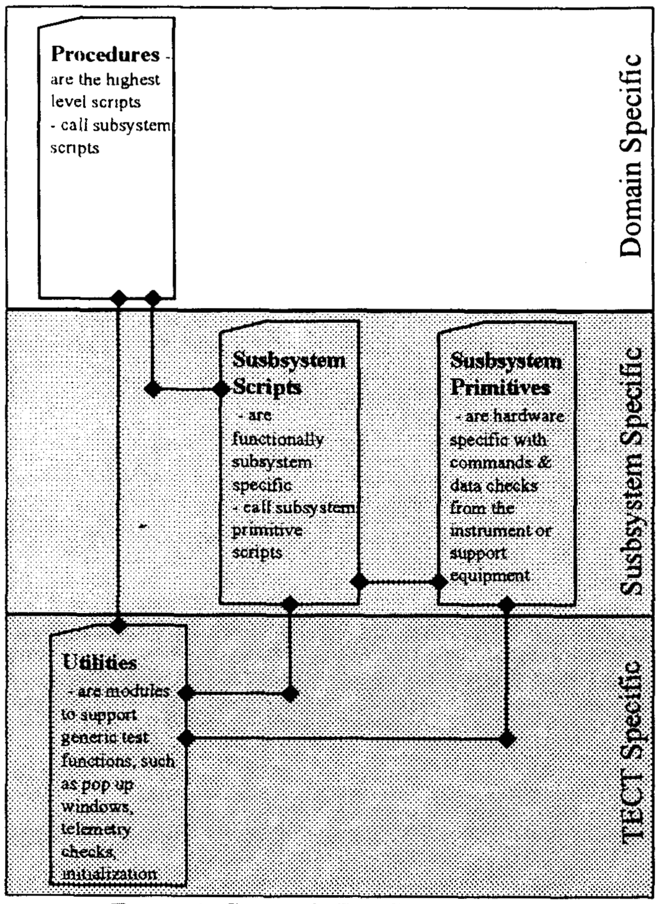

Procedures - are the highest level scripts - call subsystem scripts
Domain Specific
Susbsystem Scripts - are functionally subsystem specific - call subsystem primitive scripts
Susbsystem Primitives - are hardware specific with commands & data checks from the instrument or support equipment
Susbsystem Specific
Utilities - are modules to support generic test functions, such as pop up windows, telemetry checks, initialization
TECT Specific

Figure 6. Seawinds Test Script Layering

Procedures represent the highest level of integration and test to be performed by the script. These are system level tests and are scheduled items within the instrument test life cycle. Subsystem scripts represent the functions of a subsystem to be tested. Although they are specific to the requirements and design of the subsystem they are not tied to any one hardware or software implementation. The subsystem primitive scripts contain implementation specific commands and data to be checked from the subsystem. For control feedback, valid and invalid command counts are checked prior to sending a command then the scripts wait for either counters to increment before processing or failing on a timeout with a warning to the user or event log. Additionally, if commands generate changes to telemetry values that change is waited for or fails on a timeout with a warning to the user or event log.

The following is part of a script, called CDS_relay_control.scp. It illustrates the calling sequence and modularity between the subsystem primitive scripts and the utility scripts.

```
#---- Build command ------
set script_cmd "relay $rc_relay_name $rc_state"
set command "$Script_command,$script_cmd"

#--- An example of calling  a utility script in another file
#---- check the current command counters
if { $rc_relay_type  == $inst_relay } {
   call CDS_get_command_counts.scp
}

#---- send and log the command
set reply [route SE.CDS "$command"]

#--- An example of calling  a utility script in another file
#--- check command counters increment properly
call CDS_wait_command_count.scp

#--- if wait for command count increment times out
 if { $Command_count_status != $CDS_good_command } {
     callDisplay_error.scp
       "Script:$scriptname\nArgs:$args\nReply:$reply\nCommand count error"

. . . remainder of script removed . . .
```

Most of the variables are passed in as parameters. The parameters can determine which calls get made internal to the script. For example there is a procedure script, called Configuration.scp, where the parameters passed in are which subsystem to configure, eg. cdsa, cdsb, twta, twtb. If the parameter passed to the Configuration.scp was cdsa then a CDS subsystem script would be called, ie. CDS_select_CDS_A.scp. This subsystem script would in turn call the script, CDS_relay_control.scp shown above, controlling the appropriate relays for CDS A.

Scope of the test scripting

In order to get a better understanding of the test scripting effort it is important to get a metric for the complexity of the system under test. For the Seawinds instrument, there are approximately 111 commands, 676 telemetry points and three subsystems.

The Seawinds instrument integration and test program provides us with metrics to assess the level of effort required to automate the test program. For the Seawinds integration and test program, 115 procedure level scripts were developed. Each of these 115 procedures was supported by an additional 327 subsystem and subsystem primitive scripts and 73 common utility scripts. The subsystem scripts were roughly split among the two major subsystems of the instrument. There were 14,919 lines of Tcl written for these scripts.

5. Results and Future Work

This paper presented a test scripting tool that is used to provide high degrees of test script automation. The paper discussed the TECT functionality as it applies to the application of spacecraft and instrument test and integration. The TECT design was discussed including why scripting languages were used and the reasons for selecting Tcl as the TECT language of choice. Finally the operational aspects of test script development were discussed using the Seawinds case study.

A high degree of automation is possible and tests can be made more robust when scripted, whether or not control feedback is included. Further study of additional test cases should show that the system test procedures are highly repeatable by using script automation and that the level of effort and time required to perform testing should be reduced for the second system under test or as tests are rerun in different test environments.

Functionally TECT will be extended for the SIRTF Mission to include interfaces to mission planning and sequencing software. It will also be prototyped as a monitor and control system for the Multi-mission Ground Data System (MGDS) supporting all JPL deep space mission operations.

Finally, the TECT design will be reviewed to consider capabilities provided in newer versions of Tcl. These may include a commercial debugger as a replacement for the graphical user interface, using Tcl libraries as a replacement for the call extension and adding user community extensions to the configuration of TECT.

6. References

[1] John Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[2] G. Squibb, C. Eaton, *TMOD Roadmap; Deep Space Network Long Range Plan Consistent with NASA Code-S Mission Set*, January 28, 1998.

[3] *Seawinds Experiment Ground Support Equipment Requirements document*, JPL D-11963, December 1994.

[4] *Users Guide for the Test Execution Control Tool*, MGSO0084-00-03 (JPL D-12479), December 1995.

[5] *Scriptics, System Programming or Scripting?*, Scriptics Corporation, October 6, 1999.

[6] Cameron Laird and Kathryn Soraiz, *Chosing a scripting language*, SunWorld, October 1997.

[7] Jean-Claude Wippler, *How to use extensions in TCL*, 1998.

***Mike Levesque** is a member of the Technical staff at Jet Propulsion Laboratory. He serves as the Telecommunications Services System Service Development Engineer and performs lead system engineering for telemetry, command and data management from the space link through JPL mission control. He is a BSCS from Rensselear Polytechnic Institute. He has received the NASA Exceptional Service Medal for his contributions in developing Mission Operations Systems while at JPL.*

***John Louie and Ana Guerrero** are also members of the Technical staff at JPL.*

Acknowledgements; Dianne Fisher for her dedication to user understanding and allowing us to reproduce sections of the TECT Users Guide. Rob Gaston, John Wirth and Doug Russell for providing the Seawinds case study. Joe Diep and Rommel Mojica for making the TECT work in Seawinds ATLO. Susan Kurtik for sponsoring our effort.